社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

プレファクタリングのための不吉な臭いの検出結果の優先順位付け

セーリム　ナッタウット† 　林　　晋平† 　佐伯　元司†

† 東京工業大学 大学院情報理工学研究科 計算工学専攻
〒 152–8552 東京都目黒区大岡山 2–12–1–W8–83
E-mail: †{natthawute,hayashi,saeki}@se.cs.titech.ac.jp

あらまし　プレファクタリングの適用箇所を特定するため，ソースコード中の不吉な臭いの検出器が提案されている．しかし，既存の不吉な臭い検出器は，開発者の現在の開発コンテキストを考慮せず，関連する臭いと関連しない臭いを混在させて出力するため，特定のコンテキストに従っている開発者には適さない．その結果，開発者は適する臭いを特定するための時間を必要とするという課題がある．本稿では，不吉な臭いを開発者の持つコンテキストに従い優先順位付けする手法を提案する．我々は，イシュー管理システムに登録された，次のリリースまでに解決すべきイシューの一覧を開発のコンテキストと見なす．提案手法では，イシューの説明文に対して機能捜索手法を適用して得た結果のモジュール一覧を用いて，コンテキストに関連付く臭いを特定する．コンテキストに関連付く度合いによる優先順位に基づき，不吉な臭い検出器の出力を並べ替えて出力する．本稿では，オープンソースプロジェクトを用いて行った提案手法の予備評価についても述べる．
キーワード　リファクタリング, 機能捜索, 不吉な臭い

Toward Prioritizing Code Smell Detection Results for Prefactoring

Natthawute SAE-LIM†, Shinpei HAYASHI†, and Motoshi SAEKI†

† Department of Computer Science, Graduate School of Information Science and Engineering,
Tokyo Institute of Technology
2–12–1–W8–83, Ookayama, Meguro-Ku, Tokyo, 152–8552 Japan
E-mail: †{natthawute,hayashi,saeki}@se.cs.titech.ac.jp

Abstract　In order to find the opportunities for applying prefactoring, several techniques for detecting bad smells in source code have been proposed. However, existing smell detectors are often not suitable for developers who have a specific context because these detectors do not consider their current context and output the results that are mixed with both smells that are and are not related to such context. Consequently, the developers have to spend a considerable amount of time identifying relevant smells. In this paper, we propose a technique to prioritize bad code smells by using developers' context, i.e., a list of issues in an issue tracking system that needs to be implemented before next release. We applied feature location technique to the list of issues and used the results to specify which smells are associated with the context. Thus, our approach can provide the developers with a list of prioritized bad code smells that is related to their current context. Several preliminary evaluations using open source project indicated the effectiveness of our technique.
Key words　Refactoring, feature location, bad code smells

1. Introduction

Refactoring is a technique for improving the structure of the software without changing its functionality [1]. Code fragment that indicates problems and should be refactored is called design flaw or code smell. Many types of code smells are summarized as a smell catalog with their names [1]〜[3]. Code smells are often introduced when implementing new features or by developers with high workloads [4]. In order to suggest refactoring opportunities to developers, various types of code smell detectors have been proposed by detecting these code smells automatically [5].

According to the work by Meng et al. [6], developers do not typically refactor their code unless they have to change the code for fixing some bugs or introducing new features. However, existing code smell detectors generate a list of code smells without considering such developers' current context. Most of the existing code smell detectors analyze the specified source code and present the detected smells ordered by the attribute that is not related to the context of the developers such as module or severity to the developers. Such list of smells is inappropriate when the user developers have specific context such as the features that are planned to be implemented because both smells that are and are not related to the developers' context are mixed and scattered all over the list of smells. As a consequence, the developers have to spend a considerable amount of time specifying relevant smells that fit their context since fixing such smells may contribute to support their future implementation, i.e., improving the understandability or extendibility of the program. Such activity is time-consuming, in analogy with that static analysis tools are not used well by developers due to the large number of the detected warnings [7].

In this paper, we propose a technique to prioritize code smells from code smell detectors by considering developers' current context. This technique focuses on supporting the *prefactoring* phase [1]. In the prefactoring phase, developers improving source code's extendibility and understandability by refactoring their source code before implementing a feature to facilitate their implementation. In the proposed technique, we regard the change descriptions in an issue tracking system that are going to be implemented by a particular milestone as the developers' context. Such changes are going to be implemented by modifying or extending existing modules in the source code. These existing modules can be located by using feature location technique [8], [9]. We consider such modules as relevant modules because they are likely to be the location for implementing the new feature in the change description. As a result, code smells that appear in relevant modules should have higher priority so that the developers could easily distinguish them from irrelevant code smells. Thus, with supporting from our technique, developers can obtain a prioritized list of code smells based on the relevance to their context.

We have preliminary evaluated our technique with several open source projects, and the results indicate the potential of the effectiveness of our technique.

The main contribution of this paper is to show that the relevance between developers' context and code smells can be a useful criterion for prioritizing code smells for prefactoring. The rest of this paper is organized as follows. First, the next section explains our approach and its automated toolchain.

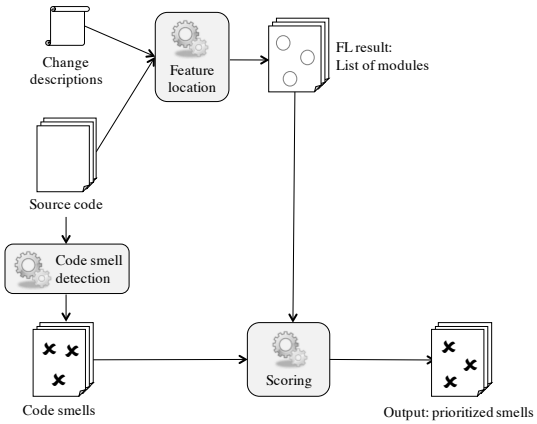


Figure 1: Overview of the proposed technique.

Section 3 discusses our preliminary evaluation. Section 4 describes related works. Finally, section 5 concludes this paper.

2. Proposed Technique

In this paper, we focus on supporting an issue-driven software development project adopting an issue tracking system such as SourceForge or Bugzilla to manage their lists of issues. Assuming that such projects have a list of issues that need to be implemented before a particular milestone, i.e., releasing major or minor versions. Such list of issues is useful to estimate developers' context, and we regard the modules that are likely to be modified as our estimation.

We propose a technique for prioritizing code smell detection results from existing code smell detectors by considering the list of issues in the issue tracking system that developers need to solve. Figure 1 shows an overview of our technique. Each gray node represents a subprocess of our technique. The input of the process is a list of issues' change descriptions obtaining from the issue tracking system and the source code of the targeted project. The output is the prioritized list of code smells based on the relevance to the developers' context. Our approach first uses the feature location technique to obtain the list of modules that are likely to be the targeted modules of each change description. Next, we generate a list of code smells by applying an existing code smell detector with the source code of the focused project. Then, for each code smell in the list, we calculate the score based on the relevance of the prior result from feature location technique. Finally, we output the prioritized list of code smells ordered by the score value mentioned above.

In our approach, we use the existing feature location and code smell detection techniques. The next subsections describe the explanation of these techniques and how we apply them.

2.1 Using Feature Location Technique

Many types of feature location techniques have been pur-

Table 1: Example portion of code smell detector results

Type	Entity	Granularity	Severity
SAP Breakers	org/gjt/sp/jedit	Subsystem	6
Blob	org.gjt.sp.jedit.Buffer	Class	7
Feature Envy	buildDOM() : void	Method	10

posed with different kinds of input and output [9]. Our approach uses the one that takes a change description d and source code C as inputs and provides a set of methods $M = \{\dots, m, \dots\}$ with their probability as outputs. The reason that we chose this type of feature location technique is that we focus on supporting the issue-based software development project which the developers tend to implement features or fix bugs by following the change description in an issue tracking system. In this situation, we assume that such change description describes the new behavior of existing feature, i.e., fixing bugs or improving functionalities. These existing features can be located by feature location technique. Therefore, the located modules of these features can be the candidates to be modified in order to achieve the change. Consequently, applying the prefactoring technique to these modules is likely to support the developers' implementation, i.e., improving understandability or extendibility of the source code. Thus, this type of feature location technique suits our needs.

In this paper, we input a set of change description $D = \{d_1, \dots, d_n\}$ and source code C to the feature location technique and obtain a series of sets of methods $\{M_1, \dots, M_n\}$.

2.2 Using Code Smell Detection

Code smell detection is a technique that generates a list of code smells from targeted source code. One example of the approaches is to detect them based on particular metric values, e.g., lines of code (LOC). The input is the source code that we want to analyze. The output is the list of smells. Each smell consists of $\langle type, entity, granularity, severity \rangle$, where *type* is the type of the detected smell, *entity* is the module having the detected smell, *granularity* is the level of code smell consisting of subsystem, class, and method, and *severity* is an integer value for representing the strength of the smell. Table 1 shows the example of code smell result from the code smell detector. For example, the second row shows the Bob smell in `org.gjt.sp.jedit.Buffer` class with severity 7. Note that we omitted the package and class name of the method level smell.

In this approach, we apply the code smell detector and obtain the list of smells S.

2.3 Scoring

To indicate the priority of each smell, we define the *score* attribute. The value of the *score* attribute is calculated by

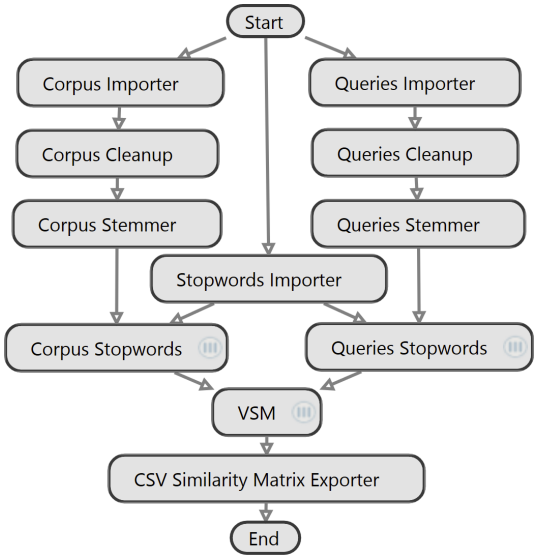


Figure 2: TraceLab configuration.

the weighted summation of the number of the modules in the result from feature location technique that match each smells' module. The *score* of each $s \in S$ can be defined as

$$score(s) = \sum_{i=1}^{n} \sum_{m \in M_i} \begin{cases} w(m), & \text{if } match(m, entity) \\ 0, & \text{otherwise} \end{cases}$$

where function $match(m, entity)$ is true when module m equals to or belongs to *entity* of each smell, and $w(m)$ is the weight of each module which can be the parameter of our technique. If we treat every module equally, $w(m)$ is equal to one. However, we can utilize the probability of each module from feature location technique result as a weight of each module. In this case, $w(m)$ becomes the probability of each module. Nevertheless, in this paper, we focus on treating every module equally. The reason will be discussed in Section 3.

2.4 Automation

We have implemented an automated tool for the proposed technique. The chain is designed to connect with an existing feature location tool. When executed, our tools trigger a code smell detector to generate a list of smells and calculate the score of each smells based on the relevance of the result from feature location technique.

For the feature location technique, we use the vector space model feature location technique [9] in a TraceLab-based solution [10] proposed by Dit et al. because vector space model is the foundation feature location technique that takes a text document such as a change description as a query and find a relevant code based on statistical methods. Figure 2 shows our configuration in TraceLab.

For smell detection, we use inFusion v1.9.0 [11] because 1) inFusion is capable of exporting the results as a file, 2) it has a command-line interface and 3) it can detect 24 types

Table 2: Data sets information

Project	Version	Size (LOC)	# Issues	# Smells
jEdit	4.2–4.3	176,098	150	387
ArgoUML	0.22–0.24	272,590	91	404

of code smells such as Blob Class, Feature Envy, These characteristics suits our approach.

2.5 Limitation

In our approach, the result is mostly affected by the accuracy of feature location technique. That is to say, if we use other types of feature location technique, the result may differ.

3. Preliminary Evaluation

We have conducted a preliminary evaluation to verify whether our approach has the potential of effectiveness with the following evaluation questions:

EQ 1: *Are relevant code smells placed in the higher rank of the list with our technique?*

The aim of our approach is to put the relevant code smells in the higher rank of the list for supporting the developers specifying relevant smell. Thus, we confirm it by conducting an experiment and analyzing the result.

EQ 2: *Does our technique applicable to any entity type?*

Since there are three types of entity generated by code smell detector: subsystem, class, and method. We applied our technique separately to each entity type and observed the result.

EQ 3: *Which weighting scheme provide a better result: treating every smell equally or using the probability value from feature location result?*

As mentioned in Section 2, when calculating the *score* value, we can either treat each module equally or use the probability value from feature location result. We did an experiment to see which is the better scheme.

3.1 Data Collection

In this evaluation, jEdit(*1) and ArgoUML(*2), active open source projects, were our subjects because their data are available through feature location benchmark [9] and we can obtain the gold set methods, the methods that were modified by developers, which were associated with a particular issue in the issue tracking system. Table 2 shows the information of our data sets including the size of the source code of the earlier version, the number of issues that we used in between two versions, and the number of smells detected by inFusion v1.9.0 [11].

We first defined the oracle as a set of code smells that occur in the modules that were modified by developers during two releases according to the data in feature location bechmark because these code smells are relevant to the developers' context as we discussed in the previous section. As for jEdit, we prepared the oracle by first applying the source code at version 4.2 to the code smell detector and obtained the result. Next, we prepared the gold set methods, the methods that were actually modified in order to solve each extractable issue in jEdit's issue tracking system by feature location benchmark during version 4.2 and 4.3. Finally, we intersected these two sets together so that we can obtain a list of smells that is actually related to the developers' context. We applied the same process to ArgoUML version 0.22 and 0.24.

As for the baseline of our evaluation, we used the original result from inFusion v1.9.0 [11] sorted by the severity of each smell.

3.2 Data Analysis

For evaluating the result of our approach, we use average precision [12] as a criterion because it is considered a reasonable metric for evaluating the quality of ranking documents. The relevant documents in the higher rank more contribute to the average precision than the relevant documents in the lower rank. Therefore, since the aim of our technique is to rearrange the result from a code smell detector and put the relevant code smells in the higher rank, the average precision of the result from our technique should be higher than the baseline of our evaluation, the original result from code smell detector. The following formula can calculate average precision.

$$AveragePrecision = \frac{\sum_r P@r}{R}$$

where r is the rank of each relevant document, R is the total number of relevant documents, and $P@r$ is the precision of the top-r retrieved documents. In this context, the relevant document is the code smell that matches the items in the oracle, and the retrieved document is the code smell in the result from the code smell detector.

We calculated the average precision of the baseline and the result from our tools ordered by the *score* of each smell for both jEdit and ArgoUML.

3.3 EQ 1

As we can see from Figure 3, the average precision of the result from our technique is significantly higher than the average precision of the baseline. This means that after prioritizing the list of smells by our technique, smells that are related to the developers' context are on the higher rank of the list. As a result, the developers can directly focus on the top rank smells without specifying which smell is or is not related to their context.

(*1) : http://www.jedit.org/

(*2) : http://argouml.tigris.org/

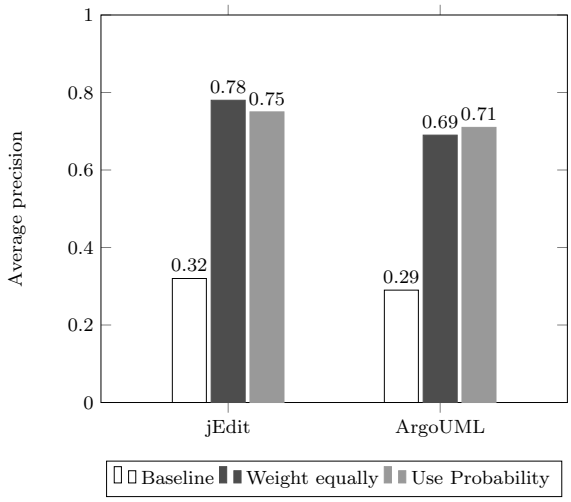


Figure 3: Comparison of the average precision value between results of baseline and our approach.

We analyzed the result by considering the 1st rank in the result from our technique. That is the Cyclic Dependency smell of package `org.argouml.uml`. This smell was ranked 123rd in the baseline. However, by applying our technique, this smell becomes the 1st rank of the list with the highest score. This is because this smell is related with many issues that need to be solved by the developers. We confirmed it by investigating the actual changes that were made during revision 0.22–0.24. We found that out of 91 issues, 65 issues were implemented in `org.argouml.uml` package which contains this CyclicDependency smell. Therefore, if the developers realized the importance of this smell and fixed it, it might be able to facilitate their implementation, i.e., improving the understandability or extendibility of source code for 65 issues. On the contrary, the 1st rank in the baseline that is the SAPBreaker smell of package `org.argouml.cognitive.checklist` is ranked 61st in the result from our technique. This is also because this smell is related with only a few issues in the issue tracking system. We also investigated the actual change and found that there is no issue implemented in `org.argouml.cognitive.checklist` package. Thus, if the developers picked the 1st smell in the original result from code smell detector and fixed it, it might not support their implementation for any issue at all.

This evidence indicates that a list of smells ordered by the relevance to developers' context has a potential be able to support developers' implementation more than the original order such as severity.

3.4 EQ 2

Figures 4a and 4b shows the average precision of method, class and subsystem entity types of jEdit and ArgoUML projects respectively. In case of jEdit project, all of the average precision for each entity type is higher with our technique. However, in ArgoUML case, only the average precision value of class and subsystem entity type is increased, not the method entity type. We investigated the result and found that there are many smells that have zero *score*, but they are matched with the items in the oracle. This means that our approach predicted that these smells are not related to the developers' context while they actually are. One of the reasons that might be the cause of this situation is the accuracy of feature location technique. Since our technique relies solely on the result of feature location technique, the accuracy of feature location technique can also affect the accuracy of our technique. That is to say, the feature location technique may have failed to locate the correct module that is the targeted of a change description. Consequently, our method incorrectly predicted that this smell is not related to the developers' context and put it in the lower rank of the list. The reason that this is not the case for subsystem and class level smells is that when we calculate the *score* value of each smell, the module m from feature location result must equal to or belongs to *entity* of each smell s. Therefore, the coarse-grained level code smells such as subsystem or class level code smells tend to satisfy the criteria more than the fine-grained level code smells like method level code smells. This indicates that our technique is more appropriate with the coarse-grained level code smells.

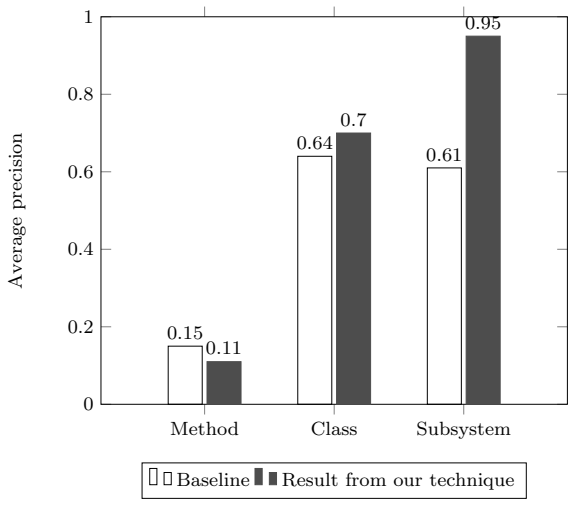
3.5 EQ 3

Figure 3 shows the result of two different weighting schemes displaying in black and grey. We can see that, for jEdit, the average precision when we treat each module equally is slightly higher than when we use the probability value as a weight. However, in case of ArgoUML, the average precision when we treat each module equally is slightly lower than when we use the probability value as a weight. Therefore, we can not determine whether which weighting scheme provide a better result.

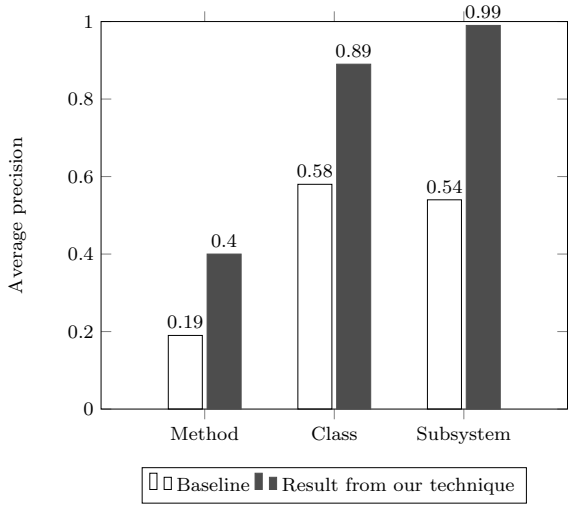
Since there is no significant difference between the two weighting schemes, we focus on the method of weighting every smells equally because of the computational cost and simplicity of the approach. However, it is possible to apply other factors to the weighting scheme such as the importance of an issue, the severity of each code smell or the effort needed for solving each code smell. This remains our future work.

4. Related Work

Some existing techniques also use the context of developers to detect code smells [13], [14], but can be regarded as supporting the *postfactoring* phase because such techniques detect code smells during the source code editing process of developers.



(a) jEdit



(b) ArgoUML

Figure 4: Comparision of the average precision value between result of baseline and our approach.

Moreover, since code smell detector tends to generate a huge number of smells, many techniques have been proposed to reduce the number of code smell detection results. Komatsuda et al. [15] proposed a technique to detect code smell that are relevant to developers' context by inserting dummy code fragment. Fontana et al. [16] proposed a technique to reduce the number of code smell detection results by applying strong and weak filters. The novel point of our approach compared to them is that our approach can be applied to any kind of smell. We prioritize every smell in the detection result based on the relevance to the developers' context.

5. Conclusion

In this paper, we proposed a technique for prioritizing code smell detection results by considering developers' current context. The result of our technique is the list of prioritized smells based on the relevance to the developers' context. The more relevant with the developers' context, the higher rank that smell is placed on the list. Therefore, our approach can assist the developers prioritizing code smells for prefactoring phase. Our technique can be used for planning how to prefactor the source code before implementing sets of issue in an issue tracking system. Our preliminary evaluation indicates that our technique is potentially useful.

Our future work includes conducting case studies to confirm that relevant code smells, as defined in this context, are useful to developers. Also, we should consider other factors that might affect developers' decision whether to fix the smells, e.g., the severity of smells, the effort needed to fix the smells or the importance of the issues. Also, More projects are needed to evaluate our technique.

Acknowledgement. This work was partly supported by JSPS Grants-in-Aid for Scientific Research (#15K15970).

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley", 1999.

[2] William C. Wake, Refactoring Workbook, Addison-Wesley, 2003.

[3] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2006.

[4] M. Tufano, F. Palomba, G. Bavota, and R. Oliveto, "When and why your code starts to smell bad," Proc. ICSE, pp.404–414, 2015.

[5] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," Journal of Software Maintenance and Evolution, vol.23, no.3, pp.179–202, 2011.

[6] N. Meng, L. Hua, M. Kim, and Kathryn S. McKinley, "Does automated refactoring obviate systematic editing?," Proc. ICSE, pp.393–402, 2015.

[7] B. Johnson, Y. Song, E. R. Murphy-Hill, and Robert W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," Proc. ICSE, pp.672–681, 2013.

[8] V. Rajlich, Software Engineering: The Current Practice, Chapman and Hall/CRC, 2011.

[9] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," Journal of Software: Evolution and Process, vol.25, no.1, pp.53–95, 2013.

[10] Center of Excellence for Software Traceability, "TraceLab," `http://www.coest.org/index.php/tracelab/`.

[11] Intooitus, "inFusion," `http://www.intooitus.com/products/infusion`.

[12] E. Zhang and Y. Zhang, "Average precision," Encyclopedia of Database Systems, pp.192–193, Springer, 2009.

[13] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting refactoring activities using histories of program modification," IEICE Transactions on Information and Systems, vol.E89-D, no.4, pp.1403–1412, 2006.

[14] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," IEEE Transactions on Software Engineering, vol.39, no.8, pp.1112–1126, 2013.

[15] T. Komatsuda, S. Hayashi, and M. Saeki, "Supporting prefactoring using feature location results," IEICE Technical Report, vol.114, no.127, pp.109–114, 2014.

[16] F.A. Fontana, V. Ferme, and M. Zanoni, "Filtering code smells detection results," Proc. ICSE, pp.803–804, 2015.